Detection of Side Channel Attacks Based on Data Tainting in Android Systems

Mariem Graa, Nora Cuppens-Boulahia, Frédéric Cuppens, Jean-Louis Lanet, Routa Moussaileb

Detection of Side Channel Attacks based on Data Tainting in Android Systems

Mariem Graa[1], Nora Cuppens-Boulahia[1], Frédéric Cuppens[1], Jean-Louis Lanet[2], Routa Moussaileb[1]

[1]IMT Atlantique, 2 Rue de la Châtaigneraie, 35576 Cesson Sévigné - France
{mariem.graa,nora.cuppens,frederic.cuppens,routa.moussaileb}@imt-atlantique.fr
[2]Campus de beaulieu, 263 Avenue Général Leclerc, 35042 Rennes - France
jean-louis.lanet@inria.fr

Abstract. Malicious third-party applications can leak personal data stored in the Android system by exploiting side channels. TaintDroid uses a dynamic taint analysis mechanism to control the manipulation of private data by third-party apps [9]. However, TaintDroid does not propagate taint in side channels. An attacker can exploit this limitation to get private data. For example, Sarwar *et al.* [2] present side channel class of attacks using a medium that might be overlooked by the taint-checking mechanism to extract sensitive data in Android system. In this paper, we enhance the TaintDroid system and we propagate taint in side channels using formal policy rules. To evaluate the effectiveness of our approach, we analyzed 100 free Android applications. We found that these applications use different side channels to transfer sensitive data. We successfully detected that 35% of them leaked private information through side channels. Also, we detected Sarwar *et al.* [2] side channel attacks. Our approach generates 9% of false positives. The overhead given by our approach is acceptable in comparison to the one obtained by TaintDroid (9% overhead).

1 Introduction

Android devices account for 80.7 % of the global smartphone sales in most markets in the world [8]. With the continuous demand of these systems, the user privacy threat is growing. Malicious applications aim to steal personal data stored in the device or potentially available through side channels such as timing, storage channels, etc. Side channel attacks [13], [18], [4] exploit the use of medium to infer private information (SMS, contacts, location, phone number, pictures...) by analyzing side channels. Sarwar *et al.* [2] proposed side channel attacks such as the bypass timing, bitmap cache, meta data, and graphical properties attacks that create taint free variables from tainted objects to circumvent the dynamic taint analysis security technique. Kim *et al.* [15] utilized screen bitmap memory attack proposed by Sarwar *et al.* to propose a collection system that retrieves sensitive information through screenshot image. The Android security model is based on application sandboxing, application signing, and a

permission framework. The side channel attack runs in its own process, with its own instance of the Dalvik virtual machine. It accesses to side channels that are a public medium. Consequently, the application sandboxing technique cannot detect these attacks. The malicious application that implements side channel attacks is digitally signed with a certificate. Therefore, it can be installed on Android systems. The standard Android permission system controls access to sensitive data but does not ensure end to end security because it does not track information flow through side channels. As the core security mechanisms of Android cannot detect side channel attacks, new approaches that extend the Android OS have been proposed. XManDroid [3], a security framework, extends the monitoring mechanism of Android to detect side channel attacks such as Soundcomber. However, it cannot detect subset of side channels such as timing channel, processor frequency and free space on filesystem. TaintDroid [9], an extension of the Android mobile phone platform, uses dynamic taint analysis to detect direct buffer attack. The dynamic analysis approach [9], [19], [11] defined in smartphones cannot detect software side channel attacks presented in [2], [15]. In this paper, we modify the Android OS to detect software side channel attacks that try to bypass detection mechanisms based on dynamic taint analysis. We propagate taint in timing, memory cache and GPU channels and in meta data (file and clipboard length) using taint propagation rules. To evaluate the effectiveness of our approach, we analyzed 100 free Android applications. We found that these applications use different side channels to transfer sensitive data. We successfully detected that 35% of them leaked private information through side channels. Also, we detected Sarwar *et al.* [2] side channel attacks. Our approach generates 9% of false positives. It has a 9% overhead with respect to the TaintDroid system. The rest of this paper is organized as follows: Section 2 presents the dynamic taint analysis mechanism and the TaintDroid approach. Section 3 describes the threat model. Section 4 presents side channel class of attacks that TaintDroid cannot detect. Section 5 describes the proposed approach. Section 6 provides implementation details. We test the effectiveness of our approach and we study our approach overhead in section 7. Section 8 describes how our approach can resist to code obfuscation attacks. We present related work about side channel attacks and countermeasures in section 9. Finally, section 10 concludes with an outline of future work.

2 Background

2.1 Dynamic Taint Analysis

The dynamic taint analysis technique is used for tracking information flows in operating systems. The principle of this mechanism is to tag some of the data in a program with a taint mark, then propagate the taint to other objects depending on this data when the program is executed. It is used primarily for vulnerability detection, protection of sensitive data, and more recently, for binary malware analysis. To detect vulnerabilities, the sensitive data must be monitored to ensure that they are sent through interfaces to the outside world. Many dynamic taint

analysis tools are based on bytecode instrumentation to analyze sensitive data [16], [6]. TaintDroid implemented similar concepts to prevent leakage of private data in Android system. We present the TaintDroid system in more details in the following section.

2.2 TaintDroid

TaintDroid improves the Android mobile phone OS to control the manipulation of users personal data in realtime by third-party applications. It analyzes application behavior to determine when privacy sensitive information is leaked. TaintDroid considers that information acquired through low-bandwidth sensors (location and accelerometer), high-bandwidth information source (microphone and camera), information databases (address books and SMS messages) and device identifiers(the phone number, SIM card identifiers (IMSI, ICC-ID), and device identifier (IMEI)) are privacy sensitive information that should be tainted. So, it uses dynamic taint analysis to track propagation of tainted data at different levels: instruction level, message-level between applications and file-level. TaintDroid defines taint sinks to detect vulnerabilities. The taint sinks present interfaces to the outside world (e.g., network interface) where tainted data are not expected to be sent. Therefore, TaintDroid, issues warning reports when the tainted data are leaked by malicious applications. One limit of TaintDroid is that it cannot propagate taint through side channels except direct memory. Therefore, it can not detect side channel attacks presented in the section 4.

3 Target Threat Model

The adversary's goal is to extract sensitive data from the Android third-party system. He/She develops a malicious application that will be executed on this system and that sends sensitive data through the network to a system which the adversary controls. We assume that the smartphone user installs the malicious application on his phone. Also, we assume that he/she uses a dynamic taint tracking system such as TaintDroid to protect his private data. So, the malicious application will be executed under this system. The adversary exploits the limitation of dynamic taint analysis mechanism that it cannot propagate taint through side channels. He/She interferes in the taint propagation level and he/she removes taint of sensitive data that should be tainted. Therefore, these data will be leaked without being detected. Next, we present different examples of side channels attacks that a dynamic taint tracking system such as TaintDroid cannot detect.

4 Side Channels Attacks

Sarwar et al. [2] present side channel class of attacks such as the bypass timing, bitmap cache, meta data, and graphical properties attacks using a medium that

might be overlooked by the taint-checking mechanism to extract sensitive data. They tested and evaluated the success rate and time of these attacks with the TaintDroid system. We are interested in these attacks because they are the most important attacks presented by siwar and al. and the other attacks are already detected [10]. We present in this section examples of these side channel attacks.

4.1 Timing Attack

The timing attack is an example of a side channel attack in which the attacker attempts to compromise a cryptosystem by analyzing the time taken to gain information about the keys. Similar concept can be used to leak tainted data when running a program with taint analysis approach. Algorithm 1 presents

Algorithm 1 Timing Attack

$X_{Tainted} \leftarrow Private\_Data$
$n \leftarrow CharToInt(X)$
$StatTime \leftarrow ReadSystemTime()$
$Sleep(n)$
$StopTime \leftarrow ReadSystemTime()$
$y \leftarrow (StopTime - StartTime)$
$Y_{Untainted} \leftarrow IntToChar(y)$
$Send\_Network\_Data(Y_{Untainted})$

the timing attack in the taint tracking system. This attack exploits the system clock which is not tainted. The sleep() function suspends the execution of the current program until the waiting period that depends on the value of a tainted variable has elapsed. Therefore, the difference in time readings before and after a waiting period indicates the value of sensitive data. This difference is not tainted because there is no taint propagation in the system clock. Consequently, it can be assigned to the taint-free output variable and leaked through the network without being detected.

4.2 Cache Memory Attack

The cache memory attack is another example of side channel attacks that can be used to extract sensitive data. This attack exploits the fact that graphical output can be obtained from cache of the currently displayed screen. Algorithm 2 presents the bitmap cache attack. The graphical widget contains the private data. The attacker successfully extracts it from the bitmap cache without any warning reports because the taint is not propagated in the cache memory. He/She sends the bitmap data to a cloud and uses the Optical Character Recognition (OCR) techniques [14] to read the value of sensitive data.

Bitmap Pixel Attack: An attacker can extract private data by exploiting bitmap cache pixels as shown in Algorithm 3. He/She modifies an arbitrarily

Algorithm 2 Bitmap Cache Attack

$X_{Tainted} \leftarrow Private\_Data$
$W \leftarrow CreateNewTextWidget()$
$B \leftarrow CreateNewBitmap()$
$WriteText(X_{Tainted} \rightarrow W)$
$B \leftarrow CaptureBitmapCache(W)$
$Y \leftarrow OpticalCharacterRecognition(B)$
$Send\_Network\_Data(Y_{Untainted})$

Algorithm 3 Bitmap Pixel Attack

$X_{Tainted} \leftarrow Private\_Data$
$B \leftarrow CreateNewBitmap()$
$SetPixel([10;10], X_{Tainted} \rightarrow B)$
$Y_{Untainted} \leftarrow GetPixel(B;[10;10])$
$Send\_Network\_Data(Y_{Untainted})$

chosen pixel to represent the private data value. Then, he/she reads the value contained in this pixel at specific coordinates.

4.3 Meta Data Attacks

Taint analysis systems such as TaintDroid associate taint to the object containing sensitive data. However, these systems do not propagate taint to object size. We present side channel attacks that exploit meta data to evade taint tracking.
File length Attack:

Algorithm 4 File Length Attack

$X_{Tainted} \leftarrow Private\_Data$
$F \leftarrow CreateNewFileHandle()$
$z \leftarrow 0$
while $z < X_{Tainted}$ **do**
  $WriteOneByte(F)$
  $z \leftarrow z + 1$
end while
$Y_{Untainted} \leftarrow ReadFileLength(F)$
$Send\_Network\_Data(Y_{Untainted})$

As the file size is not tainted, an attacker can exploit this meta data to leak sensitive data, as shown in algorithm 4. Each character in private data is represented by an arbitrary file size. One byte is written to a file until its size equals to the character private data value. Then, the attacker obtains the file size which corresponds to the sensitive data without any warning reports.
Clipboard Length Attack: An attack similar to the file length Attack can

Algorithm 5 Clipboard Length Attack

$X_{Tainted} \leftarrow Private\_Data$
$z \leftarrow 0$
while $z < X_{Tainted}$ **do**
  $WriteOneByte(Clipboard)$
  $z \leftarrow z + 1$
end while
$Y_{Untainted} \leftarrow ReadFileLength(Clipboard)$
$Send\_Network\_Data(Y_{Untainted})$

be performed if an application required a clipboard to exchange data. In the clipboard length attack, the size of the file is replaced with the size of the content of the clipboard as shown in the Algorithm 5.

4.4 Graphics Processing Unit Attacks

We are interested on a graphics processing unit class of attacks that exploits the properties of a graphical elements to evade the taint tracking mechanism.

Algorithm 6 Text Scaling Attack

$X_{Tainted} \leftarrow Private_Data$
$T \leftarrow TextViewWidget()$
$T \leftarrow SetTextScalingValue(X_{Tainted})$
$Y_{Untainted} \leftarrow GetTextScalingValue(T)$
$Send\_Network\_Data(Y_{Untainted})$

For example, in the text scaling attack presented in Algorithm 6, the attacker sets an arbitrary property of a graphical widget (the scaling) with the value of private data. Then, he/she extracts and sends this property through the network.

5 Detection of side channel attacks

Our approach is based on dynamic taint analysis to overcome side channel attacks as attacks presented in Section 4. We specify a set of formally defined rules that propagate taint in different side channels to detect leakage of sensitive data.

5.1 Timing side channel propagation rule

The timing attack exploits the system clock which is available without tainting. The attacker reads the system clock after the waiting period. We define the $Timing\_Context\_Taint$ which is activated when the argument (arg) of the $Sleep()$ function is tainted.

$$Sleep(arg) \wedge Is\ tainted(arg) \implies Activate\ (Timing\_Context\_Taint)$$

In this case, we propagate taint in timing side channel. Therefore, the system clock is tainted and the attacker cannot leak sensitive information through timing side channel.

$$\textit{Is activated(Timing\_Context\_Taint)} \implies \textit{Taint(system clock)}$$

5.2 Memory cache side channel propagation rules

The bitmap cache attack exploits the cache memory of the currently displayed screen. The attacker captures the bitmap cache of a graphical object containing private data. We define the $Bitmap\_Context\_Taint$ which is activated when the graphical object is tainted.

$$\textit{Is tainted(graphical object)} \implies \textit{Activate(Bitmap\_Context\_Taint)}$$

In this case, we propagate taint in the bitmap cache side channel and we associate taint to the bitmap object.

$$\textit{Is activated(Bitmap\_Context\_Taint)} \implies \textit{Taint(Bitmap)}$$

For the bitmap pixel attack, the attacker exploits the bitmap cache pixels that is modified to get the private data value. We define the $Pixels\_Context\_Taint$ which is activated when the argument parameter of the set pixel function is tainted. So, an arbitrarily chosen pixel is changed to represent value of the private data

$$\textit{Set pixel(arg)} \wedge \textit{Is tainted(arg)} \implies \textit{Activate (Pixels\_Context\_Taint)}$$

In this case, we assign taint to the return value of *getpixel*() function.

$$\textit{Is activated(Pixels\_Context\_Taint)} \implies \textit{Taint(return\_getpixel)}$$

By using these memory cache side channel propagation rules, the attacker cannot leak sensitive information through bitmap cache memory.

5.3 Meta data propagation rule

The meta data attacks exploit the size of the object which is available without tainting. We define the $Meta\_Data\_Context\_Taint$ which is activated when the application gets private data.

$$\textit{get\_private\_data()} \implies \textit{Activate (Meta\_Data\_Context\_Taint)}$$

In this case, we define meta data propagation rule and we associate taint to the return value of the *length*() method.

$$\textit{Is activated(Meta\_Data\_Context\_Taint)} \implies \textit{Taint(length\_object)}$$

Therefore, by applying the meta data propagation rule, the attacker cannot leak sensitive information using meta data.

5.4 GPU propagation rule

The graphics processing unit class of considered attacks exploits the properties of the graphical elements (the scaling, Text size...). The attacker sets an arbitrary property of a graphical widget to the value of private data. So, we define the *GPU_Context_Taint* which is activated when the argument parameter of the Set property function is tainted.

$$\textit{Set property(arg)} \wedge \textit{Is tainted(arg)} \implies \textit{Activate (GPU\_Context\_Taint)}$$

In this case, we assign taint to the return value of *Getproperty*() function to prevent this attack.

$$\textit{Is activated(GPU\_Context\_Taint)} \implies \textit{Taint(return\_getproperty)}$$

By using the GPU propagation rule, the attacker cannot leak sensitive information by exploiting properties of the graphical elements.

6 Implementation

We modify the TaintDroid System to implement the taint propagation rules defined in Section 5. Figure 1 presents the modified components (gray components) to detect side channel attacks in TaintDroid system. We modify the dalvik virtual machine to detect timing attacks. We implement the memory cache and the GPU propagation rules at the framework level to prevent bitmap cache and GPU class of attacks. We instrument the core libraries to associate taint to meta data.

6.1 Timing Attack Detection

The $VMThread\_sleep(constu4 * args, JValue * pResult)$ function in Dalvik virtual machine native code suspends the execution of the current thread until the value of a tainted variable has elapsed. Then, the attacker reads the system clock after the waiting period. We test the argument of $VMThread\_sleep()$ to implement the $Timing\_Context\_Taint$. We modify the $currentTimeMillis(constu4 * args, JValue * pResult)$ function in the Dalvik virtual machine native code to propagate taint in the system clock if the $Timing\_Context\_Taint$ is activated. Therefore, the difference in time readings before and after a waiting period that indicates the value of sensitive data is tainted.

6.2 Cache Memory Attack Detection

We verify if the graphical object contains a private data to implement the $GPU\_Taint\_Context$. All of the graphical objects defined in the Android framework extend View. So, we check if the view is tainted. The $getDrawingCache()$ function in the view class creates and returns a bitmap object that contains the

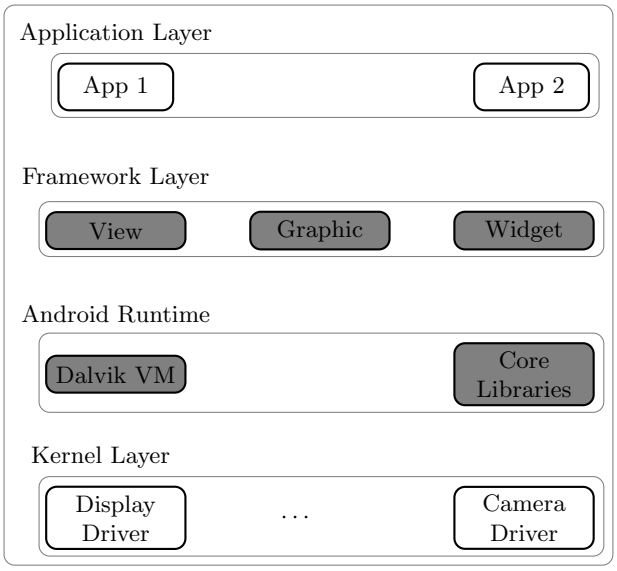


Fig. 1. The modified components (gray) to detect side channel attacks

private data. Therefore, we taint the return value of *getDrawingCache*() function if the $GPU\_Taint\_Context$ is activated. For the bitmap pixel attack, the bitmap is created in the first time and then it is modified by exploiting the bitmap cache pixels. We verify if the argument parameter of the set pixel function in Bitmap class (Graphic package) is tainted to implement $Pixels\_Taint\_Context$. In this case, we assign taint to the return value of *getpixel*() function in the bitmap class.

6.3 Meta Data Attacks Detection

TaintDroid implements taint source placement where privacy sensitive information types are acquired (low-bandwidth sensors, e.g location and accelerometer; high-bandwidth sensors, e.g., microphone and camera; information Databases, e.g address books and SMS messages; Device Identifiers, e.g SIM card identifiers (IMSI, ICC-ID), and device identifier (IMEI)). In each taint source placement, we implement the $Meta\_Data\_Context\_Taint$ wich is activated if private data is acquired. To detect the meta data class of attacks, we associate taint to the return value of the *length*() method at libcore level in File and String classes if the $Meta\_Data\_Context\_Taint$ is activated .

6.4 Graphics Processing Unit Attacks Detection

To launch the graphics processing unit class of considered attacks, the attacker sets an arbitrary property of a graphical widget with the value of private data.

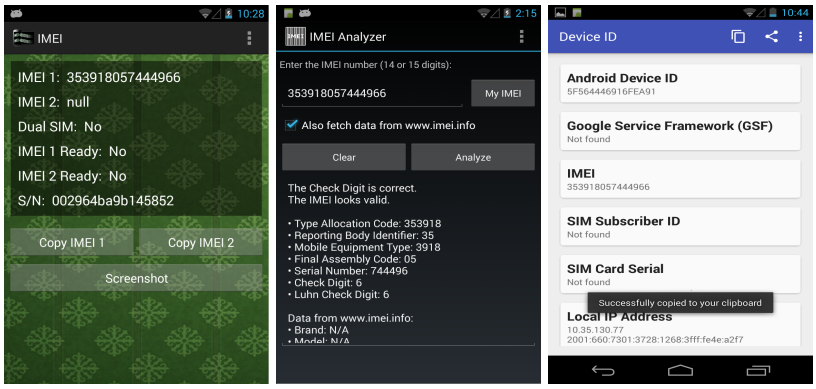
Therefore, we verify if the argument parameter of the Set-Property (*SetTextScalingValue* function) in graphical widget class is tainted to implement *GPU_Taint_Context*. Then, we taint the return value of Get-Property(*GetTextScalingValue* function) if *GPU_Taint_Context* is activated to prevent this attack.

7 Evaluation

We install our system in a Nexus 4 mobile device running Android OS version 4.3. We analyze a number of Android applications to test the effectiveness of our approach. Then, we evaluate the false positives that could occur. We study our taint tracking approach overhead using standard benchmarks.

7.1 Effectiveness

To evaluate the effectiveness of our approach, we analyze 100 most popular free Android applications downloaded from the Android Market [1]. These applications are categorized in games, shopping, device information, social, tools, weather, music and audio, maps and navigation, photograhy, productivity, life style, reference, travel, sports and entertainment applications. We observe that all applications use bitmap cache channel, 50% of these applications use timing channel, 30% use GPU channel (get and set graphic properties) and 20% use meta data (file and clipboard sizes). We found that 66% of these applications manipulate confidential data. Our approach has succesfully propagated taint in side channels and detected leakage of tainted sensitive data by checking the content of network packets sent by applications.



(a) (b) (c)

Fig. 2. Leakage private data through the bitmap cache side channels

We found that 35% of applications leaked private data through timing and bitmap cache side channels. For example, the IMEI application takes and send

a screenshot of IMEI information through the network by exploiting the bitmap cache side channel (see Figure 2 (a)). Other applications copy the SIM card and device information from the screen to the clipboard and send them through the network, SMS or bluetooth using the bitmap cache side channel (see Figure 2 (c)). Some applications get the drawing cache to leak implicitly private data. For example, the IMEI Analyser application gets the drawing cache to send implicitly the IMEI outside the smartphone (see Figure 2 (b)). Games applications leaked implicitly the devices ID through the timing side channel at the time of score sharing. In addition, we successfully implement and detect side channels class of attacks presented in Section 4.

7.2 False positives

We found that 35 of the 100 tested Android applications leaked sensitive data through side channels. We detected three device information (Android id, Device Serial, Device model, Phone number...) leakage vulnerability. Also, we detected that the IMEI is transmitted outside of smartphone by two different forms (digital and by another application which takes screenshot of IMEI). In addition, we detected four SIM card information(SIM provider's country, SIM Contacts, SimState...) leakage vulnerability. As the user is sent these information by email, SMS or bluetooth, we can not treat these applications as privacy violators. Therefore, our approach generates 9% of false positives.

7.3 Performance

We use the CaffeineMark [7] to study our approach overhead. The CaffeineMark scores roughly correlate with the number of Java instructions executed per second and do not depend significantly on the amount of memory in the system or on the speed of a computers disk drives or internet connection. Figure 3 presents the execution time results of a Java microbenchmark.

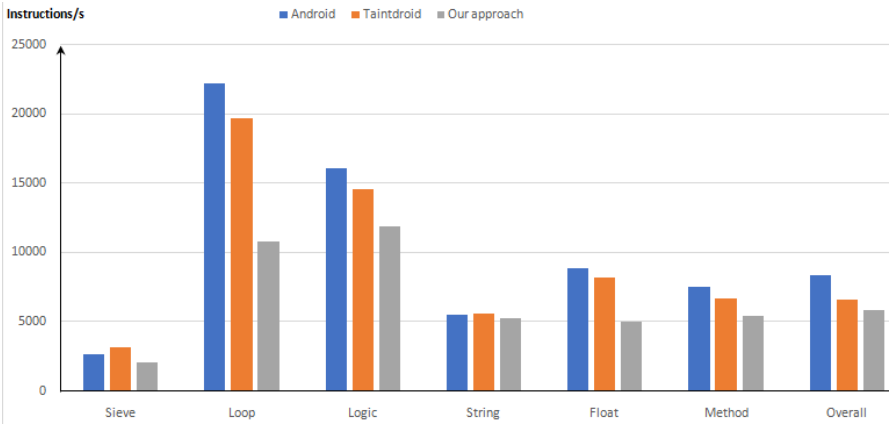


Fig. 3. Microbenchmark of Java overhead

The unmodified Android system had an overall score of 8401 Java instructions executed per second and the TaintDroid system measured 6610 Java instructions executed per second. Therefore, TaintDroid has a 21% overhead with respect to the unmodified Android system. Our approach had an overall score of 5873 Java instructions executed per second. So, our approach has a 9% overhead with respect to the TaintDroid system. It gives a slower execution speed rate because we propagate taint in side channels. However, the overhead given by our approach is acceptable in comparison to the one obtained by TaintDroid.

8 Discussion

We have proposed in a previous work [10] an enhancement of the TaintDroid approach that propagates taint along control dependencies to track implicit flows in smartphones. We have shown that our previous approach can resist to code obfuscation attacks based on control dependencies in the Android system. In addition, we have successfully detected side channel attacks exploiting control flows by combining static and dynamic analyses. However, we did not propagate taint in side channels. Consequently, we could not detect these class of attacks when they do not use control flows which generates false negatives. An attacker can obfuscate the application code by exploiting side channels to evade detection of leakage of private data in the Android system. The approach proposed in this paper propagates taint in a specific side channels. So, our approach can be used to detect code obfuscation attacks based on side channels in the Android system. We can extend our approach based on taint analysis to detect other side channel attacks such as ACCessory [17] and Soundcomber [18] attacks. To do so, we propagate taint in accelerometer and audio side channels. The limitation of our approach is that it can not be used to detect hardware side channel attacks.

9 Related work

In this section, we present side channels attacks in Android systems. We also discuss existing countermeasures.

9.1 Software Side channels Attacks

Memento [13] is a side-channel attack based on tracking changes in the browser's memory footprint to infer which pages the victim is browsing. Soundcomber [18] analyzes audio side channel in a user's phone conversations to infer sensitive data. TouchLogger [4] exploits different vibrations when typing on different locations on the touch screen to extract sequences of entered text on smartphones. ACCessory [17] uses the accelerometer to get the data entered by user. Chen *et al.* [5] exploit a shared-memory side channel to stealthily inject into the foreground a phishing activity and steal sensitive information. Kim *et al.* [15] utilized screen bitmap memory attack proposed by Sarwar *et al.* [2] to propose a collection system that retrieves IMEI and IMSI information through screenshot images. As

we propagate taint in bitmap cache memory, we can detect Kim *et al.* attacks. We are interested on software side channel attacks that try to bypass dynamic taint analysis based detection technique such as timing, memory cache, GPU channels and meta data (file and clipboard length) [2], [15].

9.2 Side channels Countermeasures

Many works exist in the literature to detect side channel attacks in Android systems. App Guardian [20] thwarts a malicious apps runtime monitoring attempt by pausing all suspicious background processes, which are identified by their behaviors inferred from their side channels. In this paper, we are interested in side channel attacks running in the foreground. So, App Guardian cannot detect this category of attacks. XManDroid [3] uses dynamic taint analyses to detect side channel attacks such as Soundcomber. However, it cannot detect subset of side channels such as timing channel, processor frequency and free space on filesystem. TaintDroid [9] uses dynamic taint analysis to detect direct buffer attack. DroidBox [19] analyzes the malicious applications by using sandbox and tainting techniques based on TaintDroid. It combines static and dynamic analysis, and it uses machine learning techniques to cluster the analyzed samples into benign and malicious ones. AppFence [11] extends TaintDroid to implement enforcement policies. The dynamic analysis approach defined in smartphones like TaintDroid, AppFence and DroidBox cannot detect software side channel attacks presented in [2], [15].

10 Conclusion

The dynamic taint analysis approaches implemented in the Android system can be bypassed by exploiting side channel attacks. We have improved the TaintDroid approach to propagate taint in side channels. We have analyzed 100 free Android applications to evaluate the effectiveness of our approach. We successfully detected sensitive information leakage caused by side channels. We found that 35% of analyzed applications leaked private data through side channels. We showed that our approach generates significant false positives (9%). Our approach creates a 9% overhead with respect to the TaintDroid system. Future work will be to improve our approach for detecting other side channel attacks inferring private data. Also, we will demonstrate the completeness of the taint propagation rules.

References

1. Android market, https://play.google.com/store/apps?hl=fr
2. Babil, G.S., Mehani, O., Boreli, R., Kaafar, M.A.: On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In: Security and Cryptography (SECRYPT), 2013 International Conference on. pp. 1–8. IEEE (2013)

3. Bugiel, S., Davi, L., Dmitrienko, A., Fischer, T., Sadeghi, A.R.: Xmandroid: A new android evolution to mitigate privilege escalation attacks. Technische Universität Darmstadt, Technical Report TR-2011-04 (2011)
4. Cai, L., Chen, H.: Touchlogger: Inferring keystrokes on touch screen from smartphone motion. HotSec 11, 9–9 (2011)
5. Chen, Q.A., Qian, Z., Mao, Z.M.: Peeking into your app without actually seeing it: Ui state inference and novel android attacks. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 1037–1052 (2014)
6. Cheng, W., Zhao, Q., Yu, B., Hiroshige, S.: Tainttrace: Efficient flow tracing with dynamic binary rewriting. In: ISCC'06. Proceedings. 11th IEEE Symposium on. pp. 749–754. IEEE (2006)
7. Corporation, P.S.: Caffeinemark 3.0, http://www.benchmarkhq.ru/cm30/ (1997)
8. Egham: Gartner says worldwide smartphone sales grew 3.9 percent in first quarter of 2016 (May 2016), http://www.gartner.com/newsroom/id/3323017
9. Enck, W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J., McDaniel, P., Sheth, A.N.: Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. ACM Transactions on Computer Systems (TOCS) 32(2), 5 (2014)
10. Graa, M., Boulahia, N.C., Cuppens, F., Cavalliy, A.: Protection against code obfuscation attacks based on control dependencies in android systems. In: Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on. pp. 149–157. IEEE (2014)
11. Hornyack, P., Han, S., Jung, J., Schechter, S., Wetherall, D.: These aren't the droids you're looking for: retrofitting android to protect data from imperious applications. In: Proceedings of the 18th ACM conference on Computer and communications security. pp. 639–652. ACM (2011)
12. Hund, R., Willems, C., Holz, T.: Practical timing side channel attacks against kernel space aslr. In: Proceedings of the 2013 IEEE Symposium on Security and Privacy. pp. 191–205. SP '13, IEEE Computer Society, Washington, DC, USA (2013)
13. Jana, S., Shmatikov, V.: Memento: Learning secrets from process footprints. In: 2012 IEEE Symposium on Security and Privacy. pp. 143–157. IEEE (2012)
14. Kay, A.: Tesseract: an open-source optical character recognition engine. Linux Journal 2007(159), 2 (2007)
15. Kim, Y.k., Yoon, H.J., Lee, M.H.: Stealthy information leakage from android smartphone through screenshot and ocr. In: International Conference on Chemical, Material and Food Engineering. Atlantis Press (2015)
16. Newsome, J., Song, D.: Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. Citeseer (2005)
17. Owusu, E., Han, J., Das, S., Perrig, A., Zhang, J.: Accessory: password inference using accelerometers on smartphones. In: Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications. p. 9. ACM (2012)
18. Schlegel, R., Zhang, K., Zhou, X.y., Intwala, M., Kapadia, A., Wang, X.: Soundcomber: A stealthy and context-aware sound trojan for smartphones. In: NDSS. vol. 11, pp. 17–33 (2011)
19. Spreitzenbarth, M., Schreck, T., Echtler, F., Arp, D., Hoffmann, J.: Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. International Journal of Information Security 14(2), 141–153 (2015)
20. Zhang, N., Yuan, K., Naveed, M., Zhou, X., Wang, X.: Leave me alone: App-level protection against runtime information gathering on android. In: 2015 IEEE Symposium on Security and Privacy. pp. 915–930. IEEE (2015)